Differentiable 3D Triangle-Triangle Intersection Energy

TIANYU WANG

Obtaining intersection-freeness or global injectivity is important in computer graphics. However, it is challenging, especially for the non-oriented deformation primitives. Most methods often rely on an intersection-free initialization and track the continuous trajectory to keep the legitimacy and cannot be used for the task without such an initialization. For the latter one in 3D space, we introduce a novel second-order differentiable energy defined from the 3D triangle-triangle intersection testing, and a GPU-based inexact Newton optimization route. We show that intersection can be efficiently resolved integrated with our method, requiring no user interaction, history information or a valid initialization.

CCS Concepts: • **Computing methodologies** → **Physical simulation**.

Additional Key Words and Phrases: collision handling, computational geometry, GPU computation, nonlinear optimization

1 3D TRIANGLE-TRIANGLE INTERSECTION ENERGY

Our method is built on the Möller's method [Möller 1997]. The key observation is that we can reduce the 3D triangle-triangle intersection test to the overlapping test of two 1D line segments. Therefore, it is possible to define a simple closed-form differentiable energy to resolve this overlapping. Once the overlapping is resolved, the triangle-triangle intersection should also be resolved.

We consider the general situation where two intersected triangles T_a and T_b are not coplanar or topologically adjacent as shown in Fig. 1: $\mathbf{u}_{0,1,2}$ are the three vertices of T_a and $\mathbf{v}_{0,1,2}$ are the three vertices of T_b; T_a intersects with the plane of T_b and forms a line segment whose parameter interval I_a on the straight line $L(t)$ is $[t_a^0, t_a^1]$; T_b intersects with the plane of T_a and forms a line segment whose parameter interval I_b on the straight line $L(t)$ is $[t_b^0, t_b^1]$. Möller showed that T_a intersects with T_b if and only if $I_a \cap I_b \neq \emptyset$. This can be further equivalent to that:

$$0 \in I_c \coloneqq [t_c^0, t_c^1] = I_a \oplus (-I_b) = [t_a^0 - t_b^1, t_a^1 - t_b^0], \tag{1}$$

as illustrated in [Minarčík et al. 2024; Schneider 2013], where $I_a \oplus (-I_b)$ means the *Minkowski sum* of the point set I_a and the negated point set of I_b. To ignore the critical situation first, we can say that T_a intersects with T_b if and only if $0 \in (t_c^0, t_c^1)$, i.e., $(0 - \frac{t_c^0+t_c^1}{2})^2 < (\frac{t_c^1-t_c^0}{2})^2$. Therefore, we define our triangle-triangle intersection energy $\mathcal{T}(\mathbf{u}, \mathbf{v})$ as below:

$$\mathcal{T}(\mathbf{u}, \mathbf{v}) = ((0 - \frac{t_c^0 + t_c^1}{2})^2 - (\frac{t_c^1 - t_c^0}{2})^2)^2 = (t_c^0(\mathbf{u}, \mathbf{v}) t_c^1(\mathbf{u}, \mathbf{v}))^2. \tag{2}$$

1.1 Gradient and Hessian

To compute the gradient of $\mathcal{T}(\mathbf{u}, \mathbf{v})$ w.r.t. $\mathbf{u}, \mathbf{v}$ is straight forward:

$$\frac{\partial \mathcal{T}(\mathbf{u}, \mathbf{v})}{\partial (\mathbf{u}, \mathbf{v})} = 2t_c^0 (t_c^1)^2 \frac{\partial t_c^0(\mathbf{u}, \mathbf{v})}{\partial (\mathbf{u}, \mathbf{v})} + 2t_c^1 (t_c^0)^2 \frac{\partial t_c^1(\mathbf{u}, \mathbf{v})}{\partial (\mathbf{u}, \mathbf{v})}. \tag{3}$$

Author's address: Tianyu Wang.

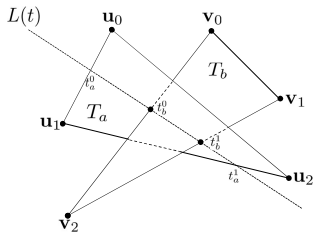

Fig. 1. **Möller's method for triangle-triangle intersection test in 3D.**

Ignoring the asymmetric part of the exact Hessian, its Hessian matrix can be expressed as below:

$$\begin{cases} \frac{\partial^2 \mathcal{T}(\mathbf{u}, \mathbf{v})}{\partial (\mathbf{u}, \mathbf{v})^2} \approx \\ 2(t_c^1)^2 \frac{\partial t_c^0(\mathbf{u}, \mathbf{v})}{\partial (\mathbf{u}, \mathbf{v})} \otimes \frac{\partial t_c^0(\mathbf{u}, \mathbf{v})}{\partial (\mathbf{u}, \mathbf{v})} + \\ 2(t_c^0)^2 \frac{\partial t_c^1(\mathbf{u}, \mathbf{v})}{\partial (\mathbf{u}, \mathbf{v})} \otimes \frac{\partial t_c^1(\mathbf{u}, \mathbf{v})}{\partial (\mathbf{u}, \mathbf{v})} + \\ 4t_c^0 t_c^1 (\frac{\partial t_c^1(\mathbf{u}, \mathbf{v})}{\partial (\mathbf{u}, \mathbf{v})} \otimes \frac{\partial t_c^0(\mathbf{u}, \mathbf{v})}{\partial (\mathbf{u}, \mathbf{v})} + \frac{\partial t_c^0(\mathbf{u}, \mathbf{v})}{\partial (\mathbf{u}, \mathbf{v})} \otimes \frac{\partial t_c^1(\mathbf{u}, \mathbf{v})}{\partial (\mathbf{u}, \mathbf{v})}), \end{cases} \tag{4}$$

where $\otimes$ is the outer product symbol.

Eigenanalysis. We further analyze the eigensystem of Eq. 4. The inexact Hessian is composed of three parts: the first part has a non-zero eigenvalue, which equals $2(t_c^1)^2 \left\| \frac{\partial t_c^0(\mathbf{u}, \mathbf{v})}{\partial (\mathbf{u}, \mathbf{v})} \right\|^2$ and is always positive; the second part has a non-zero eigenvalue, which equals $2(t_c^0)^2 \left\| \frac{\partial t_c^1(\mathbf{u}, \mathbf{v})}{\partial (\mathbf{u}, \mathbf{v})} \right\|^2$ and is always positive; the third part has a non-zero eigenvalue, which equals $8t_c^0 t_c^1 \frac{\partial t_c^0(\mathbf{u}, \mathbf{v})}{\partial (\mathbf{u}, \mathbf{v})} \cdot \frac{\partial t_c^1(\mathbf{u}, \mathbf{v})}{\partial (\mathbf{u}, \mathbf{v})}$ and might be negative.

1.2 GPU Implementation

The gradient and Hessian computation of our energy is built on an open-sourced implementation [Eberly 2024; Schneider and Eberly 2002] of Möller's method. Considering the performance and memory consumption, we do not use forward-mode or reverse-mode automatic differentiation techniques, but manually construct the computation graph to reduce the number of multiplications, and then record the related gradient to construct the final Jacobian matrix on GPU. A C++ version of the gradient and Hessian computation based on the autodiff library [Leal 2018] can be found in Appendix A for clarity. To accelerate the triangle-triangle intersection testing, we use spatial hashing structure [Pabst et al. 2010] to eliminate redundant computations.

REFERENCES

David Eberly. 2024. The Triangle-Triangle Intersection Implemented by Geometictools.com. https://www.geometrictools.com/GTE/Samples/Intersection/AllPairsTriangles/TriangleIntersection.cpp

Allan M. M. Leal. 2018. autodiff, a modern, fast and expressive C++ library for automatic differentiation. `https://autodiff.github.io`. https://autodiff.github.io

Jiří Minarčík, Sam Estep, Wode Ni, and Keenan Crane. 2024. Minkowski Penalties: Robust Differentiable Constraint Enforcement for Vector Graphics. In *ACM SIGGRAPH 2024 Conference Papers (SIGGRAPH '24)*. Association for Computing Machinery, New York, NY, USA, Article 2, 12 pages. https://doi.org/10.1145/3641519.3657495

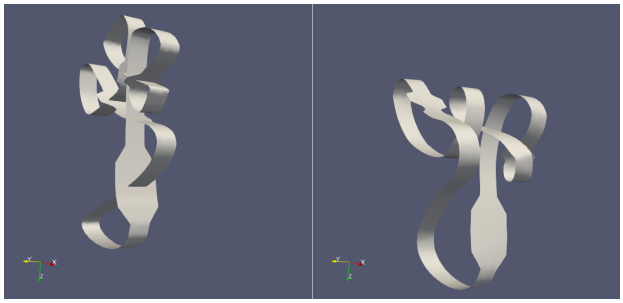

Fig. 2. **A quasi-static simulation with our 3D triangle-triangle intersection energy, which evolves from an invalid state (left) to a valid one (right).**

Tomas Möller. 1997. A fast triangle-triangle intersection test. *Journal of graphics tools* 2, 2 (1997), 25–30.

Simon Pabst, Artur Koch, and Wolfgang Straßer. 2010. Fast and Scalable CPU/GPU Collision Detection for Rigid and Deformable Surfaces. *Computer Graphics Forum* 29, 5 (2010), 1605–1612. https://doi.org/10.1111/j.1467-8659.2010.01769.x arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2010.01769.x

Philip Schneider and David H Eberly. 2002. *Geometric tools for computer graphics*. Elsevier.

Rolf Schneider. 2013. *Convex bodies: the Brunn–Minkowski theory*. Vol. 151. Cambridge university press.

A C++ CODE FOR OUR 3DTTI ENERGY

```
bool intersects(
        Vector3dual& U0, Vector3dual& U1, Vector3dual& U2,
        Vector3dual& V0, Vector3dual& V1, Vector3dual& V2,
        Vector3dual& segment0, Vector3dual& segment1) {
  Vector3dual edge1 = U1 - U0;
  Vector3dual edge2 = U2 - U0;
  Vector3dual normal = edge1.cross(edge2);
  normal.normalize();
  dual d[3];
  int positive = 0, negative = 0, zero = 0;
  d[0] = normal.dot(V0 - U0);
  d[1] = normal.dot(V1 - U1);
  d[2] = normal.dot(V2 - U2);
  for (int i = 0; i < 3; ++i) {
    if (d[i].val > 0.0) {
      ++positive;
    } else if (d[i].val < 0.0) {
      ++negative;
    } else {
      ++zero;
    }
  }
  // positive + negative + zero == 3
  if (positive > 0 && negative > 0) {
    if (positive == 2)  // and negative == 1
    {
      if (d[0].val < 0.0) {
        segment0 = (d[1] * V0 - d[0] * V1) / (d[1] - d[0]);
        segment1 = (d[2] * V0 - d[0] * V2) / (d[2] - d[0]);
      } else if (d[1].val < 0.0) {
        segment0 = (d[0] * V1 - d[1] * V0) / (d[0] - d[1]);
        segment1 = (d[2] * V1 - d[1] * V2) / (d[2] - d[1]);
      } else  // d[2].val < 0.0
      {
        segment0 = (d[0] * V2 - d[2] * V0) / (d[0] - d[2]);
        segment1 = (d[1] * V2 - d[2] * V1) / (d[1] - d[2]);
      }
    } else if (negative == 2)  // and positive == 1
    {
      if (d[0].val > 0.0) {
        segment0 = (d[1] * V0 - d[0] * V1) / (d[1] - d[0]);
        segment1 = (d[2] * V0 - d[0] * V2) / (d[2] - d[0]);
      } else if (d[1].val > 0.0) {
        segment0 = (d[0] * V1 - d[1] * V0) / (d[0] - d[1]);
        segment1 = (d[2] * V1 - d[1] * V2) / (d[2] - d[1]);
      } else  // d[2].val > 0.0
      {
        segment0 = (d[0] * V2 - d[2] * V0) / (d[0] - d[2]);
        segment1 = (d[1] * V2 - d[2] * V1) / (d[1] - d[2]);
      }
    } else  // positive == 1, negative == 1, zero == 1
    {
      if (d[0].val == 0.0) {
        segment0 = V0;
        segment1 = (d[2] * V1 - d[1] * V2) / (d[2] - d[1]);
      } else if (d[1].val == 0.0) {
        segment0 = V1;
        segment1 = (d[0] * V2 - d[2] * V0) / (d[0] - d[2]);
      } else  // d[2].val == 0.0
      {
        segment0 = V2;
        segment1 = (d[1] * V0 - d[0] * V1) / (d[1] - d[0]);
      }
    }
    return true;
  }
  return false;
}
Vector3dual 3DTTIT(Vector3dual loc, bool& intersected) {
  Vector3dual res(2);
  Vector3dual V0 = loc.segment(0, 3);
  Vector3dual V1 = loc.segment(3, 3);
  Vector3dual V2 = loc.segment(6, 3);
  Vector3dual U0 = loc.segment(9, 3);
  Vector3dual U1 = loc.segment(12, 3);
  Vector3dual U2 = loc.segment(15, 3);
  Vector3dual S00;
  Vector3dual S01;
  Vector3dual S10;
  Vector3dual S11;
  if (intersects(V0, V1, V2, U0, U1, U2, S00, S01) &&
      intersects(U0, U1, U2, V0, V1, V2, S10, S11)) {
    Vector3dual uNormal;
    uNormal = (U1 - U0).cross(U2 - U0);
    Vector3dual vNormal;
    vNormal = (V1 - V0).cross(V2 - V0);
    Vector3dual D;
    D = uNormal.cross(vNormal);
    D.normalize();
    Vector3dual A;
    A = 0.25 * (S00 + S01 + S10 + S11);
    dual t00 = D.dot(S00 - A);
    dual t01 = D.dot(S01 - A);
    dual t10 = D.dot(S10 - A);
    dual t11 = D.dot(S11 - A);

    dual ta0 = min(t00, t01);
    dual ta1 = max(t00, t01);
    dual tb0 = min(t10, t11);
    dual tb1 = max(t10, t11);

    intersected = (ta1.val > tb0.val && ta0.val < tb1.val);

    res(0) = (ta0 - tb1) * 0.5;  // tc0
    res(1) = (ta1 - tb0) * 0.5;  // tc1
    return res;
  }
  intersected = false;
  return res;
}
```